Algorithmique et UML

CM4 : Java « avancé »

Mickaël Martin Nevot

V1.13.0

Algorithmique et UML

Pile/tas

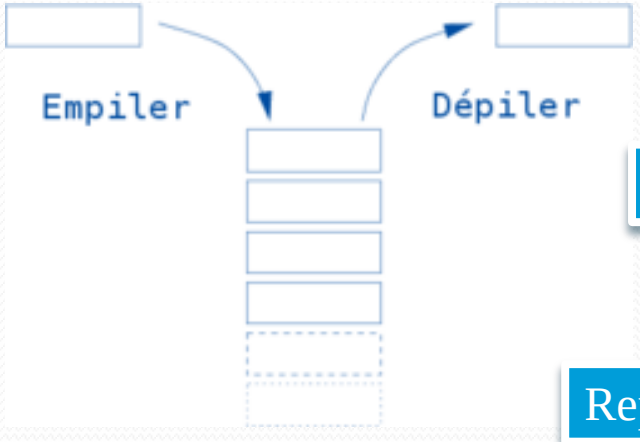
Fonctionnement dépendant de l'implémentation de la JVM : seul le principe général est expliqué ici !

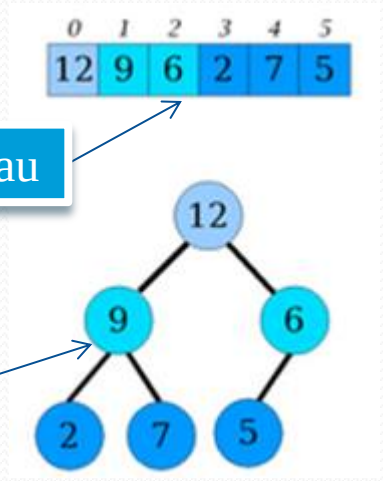
Pile (ou registre)

- Principe d'une pile :
  - Sommet : dernier élément
- Appels de méthodes
- Variables locales

Tas

- Objet :
  - Place allouée par `new`
  - Contient attributs/méthodes
  - $taille_{objet} = \sum taille_{attributs}$
  - *Garbage collector*





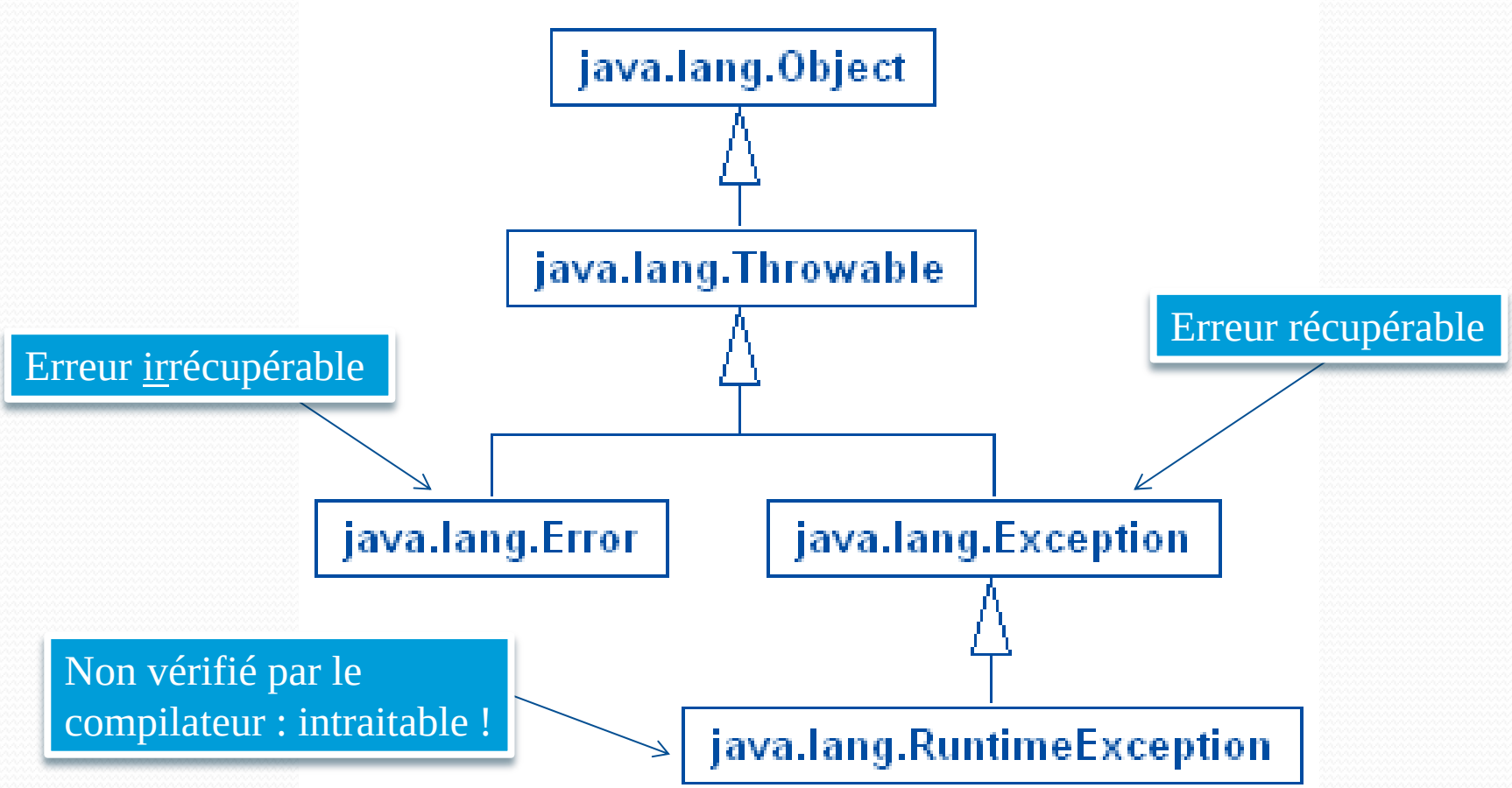
Représentation en tableau

Représentation en arbre

Exception

- Objet (sous-classe de `java.lang.Throwable`)
- Signal indiquant un cas exceptionnel :
  - Erreur : irrécupérable (arrêt de l'application)
  - Exception : récupérable (traitable)
- Interrompt le flot d'exécutions normales
- Traitement d'erreur :
  - Séparation du code normal/exceptionnel (lisibilité)
  - Récupération à un autre niveau (propagation dans la pile)
- Si propagée jusqu'en haut de la pile : arrêt de l'application
- `RuntimeException` (implicite) : non traitable

Exception : hiérarchie objet



Exceptions : modélisation

- Attributs :
  - `String message` ← Contient le message de description de l'exception
- Méthodes :
  - `Exception()`
  - `Exception(String)`

  } Constructeur avec ou sans paramètre
  - `getMessage():String` ← Renvoie `message`
  - `printStackTrace()` ← Affiche la liste des appels de méthode ayant conduit à l'exception

Exception : mots clefs

- Mot clef `try` : délimitation « d'usabilité » d'exceptions
- Mot clef `catch` : capturer l'exception (traitement)
- Mot clef `finally` : traiter les erreurs non traitées
- Mot clef `throw` : lancer l'exception (signalement)
- Mot clef `throws` : lancement d'exceptions possible

Il n'y a pas de cas d'exceptions implicites

Exceptions : exemple

1
```
// Classe d'exception simple.
public class MyException extends Exception { … }
```

2
```
…
public MyClass() throws MyException {
    // N'importe quelle méthode peut lancer des
    // exceptions, y compris un constructeur.
    …
    throw new MyException(val1); // Utilisation optionnelle d'arguments.
}
```

3
```
…
public static void main(String[] argv) {
    try {
        new MyClass();
    } catch (MyException e) { // L'ordre des blocs a une importance.
        e.printStrackTrace(); // Équivalent à : System.out.println(e.getMessage()).
    } finally {
        … // On traite ici toutes les exceptions explicites ou implicites
    }     // non traitées jusqu'ici.
}
```

Flux

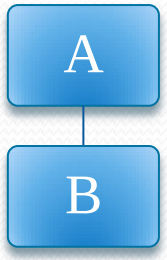
- Flux de données (comme un film en « streaming » !)
- Paquetage `java.io` :
  - Flux binaires (lecture/écriture d'octets) :
    - `InputStream`, etc.
    - `OutputStream`, etc.
  - Flux de caractères (lecture/écriture de caractères) :
    - `Reader` (`BufferedReader`, `FileReader`, etc.)
    - `Writer` (`BufferedWriter`, `FileWriter`, etc.)

```
// Lit l'entrée standard.
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
// Lit la ligne jusqu'au prochain retour chariot.
String inputLine = br.readLine();
```

Type et polymorphisme

- **Type** :
  - Défini les valeurs qu'une donnée peut prendre
  - Défini les opérateurs qui peuvent lui être appliqués
  - Défini la syntaxe : « comment l'appeler ? »
  - Défini la sémantique : « qu'est ce qu'il fait ? »
  - Une classe est un type (composé), une interface aussi…
- **Polymorphisme** :
  - Capacité d'un objet à avoir plusieurs types
  - Permet d'utiliser une classe héritière comme une classe héritée

Polymorphisme



- Surclassement (à la compilation) :
  - Vu comme un objet du type de la **référence**
  - Fonctionnalités restreintes à celles du type de la **référence**

```
A myObj = new B( … );
```

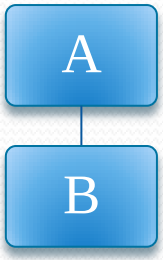
- Liaison dynamique (à l'exécution) :
  - Méthode de la **classe effective** de l'objet qui est exécutée

```
myObj.meth1( … );
```

- *Downcasting* :
  - Libère les fonctionnalités restreintes par le surclassement

```
((B) myObj).meth2( … );
```

Polymorphisme



- Surclassement (à la compilation) :
  - Vu comme un objet du type de la **référence**
  - Fonctionnalités restreintes à celles du type de la **référence**

```
A myObj = new B( … );
```

`A` : référence
`B` : classe effective

- Liaison dynamique (à l'exécution) :
  - Méthode de la **classe effective** de l'objet qui est exécutée

```
myObj.meth1( … );
```

`meth1(…)` est une méthode de `A`, redéfinie par `B` : c'est celle de `B` qui est exécutée !

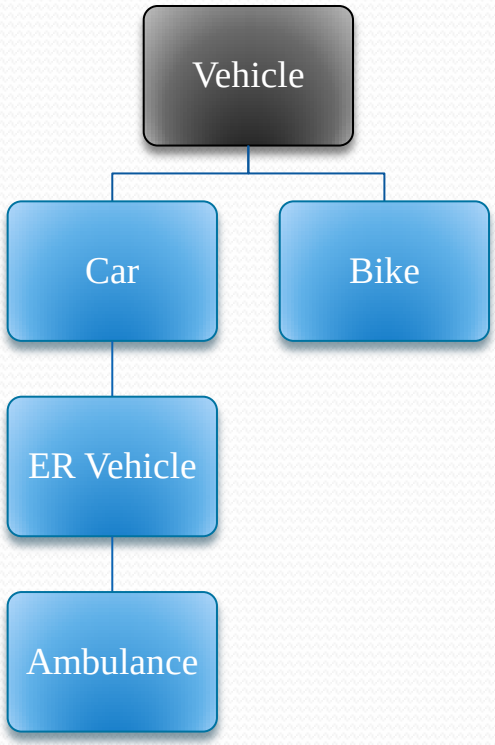
- *Downcasting* :
  - Libère les fonctionnalités restreintes par le surclassement

```
((B) myObj).meth2( … );
```

Transtypage

Polymorphisme

```
public class Vehicle {
    …
    void move() { System.out.println("Avec deux ou quatre roues !"); }
}
…
public class Bike extends Vehicle {
    …
    void move() { System.out.println("Avec deux roues !"); }
    void lean() { System.out.println("Je me penche !"); }
}
…
public static void main(String[] args) {
    Vehicle bike = new Bike( … ); // Surclassement.
    bike.move();  // Liaison dynamique.
                  // Affichage : Avec deux roues !
    bike.lean();  // Erreur !
                  //(Vehicle n'a pas de méthode lean()).
    ((Bike) bike).lean(); // Downcasting.
                          // Affichage : Je me penche !
}
```



Classe abstraite

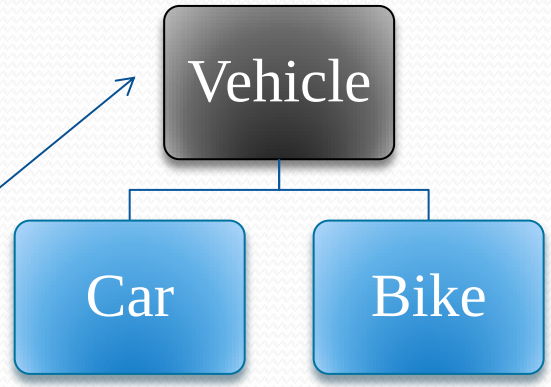
- Ne peut pas être instanciée (mais constructeur[s] possible[s])
- **Si une seule méthode est abstraite, la classe l'est aussi**
- Abstraction possible à plusieurs niveaux d'héritage
- Méthodes : accessibilité `private` impossible
- Mot clef `abstract` :
  - Classe :

    ```
    public abstract class MyClass { … }
    ```

    Pas de corps
  - Méthode :

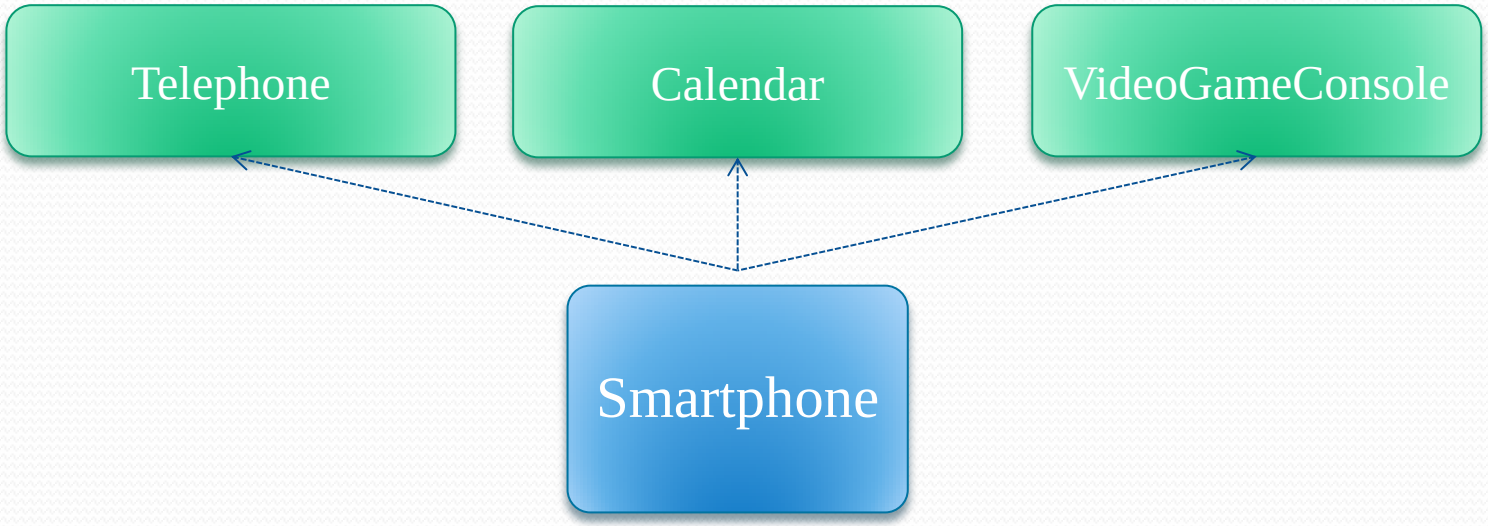
    ```
    public abstract void meth1( … );
    ```

La classe `Vehicle` est abstraite :
il n'y a pas d'instance de `Vehicle` mais
des instances de `Car` ou de `Bike`



Interface

- Modèle pour une classe
- **Classe totalement abstraite** sans attribut (non constant)
- Une classe implémentant une interface doit implanter ***(implémenter)*** toutes les méthodes déclarées par l'interface



Interface

- Une interface **donne son type** aux classes l'implémentant
- Mot clef `interface` (pas `abstract`) :

```
public interface MyInterface { … };
```

- Mot clef `implements` :

```
public class MyClass implements MyInterface1 { … }
public class MyClass1 implements MyInterface1, MyInterface2 … { … }
public class MyClass2 extends MySuperClass implements MyInterface1 … { … }
```

- Les interfaces peuvent se dériver (mot clef `extends`)

```
public interface MyInterface3 extends MyInterface1 { … };
public interface MyInterface4 extends MyInterface1, MyInterface2 { … };
```

Classes interne/anonyme

- Classe locale ou **interne** :

```
public class MyClass {
    …
    class MyLocalClass { … }
}
```

- Classe **anomyme** :

```
MyAnonymousClass myObj = new MyAnonymousClass() {
    …
};
```

Attention : il s'agit d'une instruction !

- *Bytecode* :
  - Classe : `MyClass.class`
  - Interne : `MyClass$MyLocalClass.class`
  - Anonyme : `MyClass$1.class`

Sérialisation

- Interface `Serialisable`
- Mot clef `transient` (pas de sérialisation) :

```
public class MyClass implements Serialisable {
    …
    protected MyClass2 myObj1 = new MyClass2();
    // myObj2 ne sera pas sérialisé.
    transient MyClass3  myObj2 = new MyClass3();
}
```

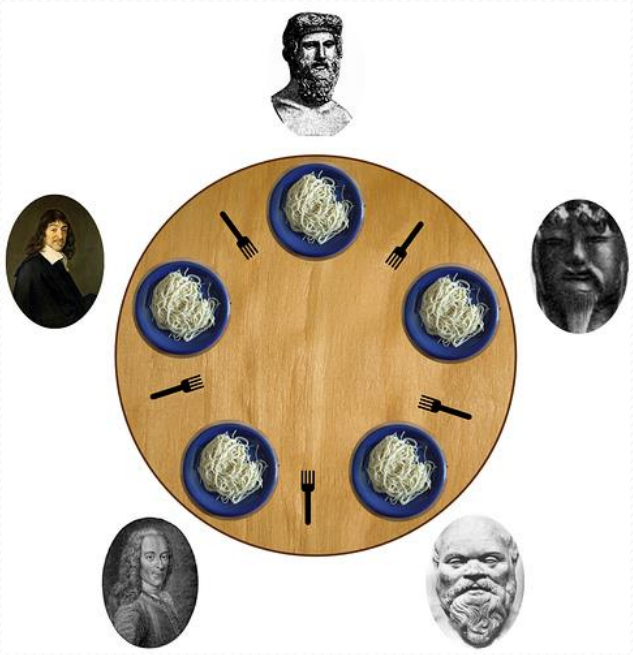
- « Désérialisation » :

```
FileInputStream fis = new FileInputStream ("myFile.ser");
ObjectInputStream ois = new ObjectInputStream (fis);
Object first = ois.readObject ();
MyClass myObj = (MyClass) first;
ois.close();
…
```

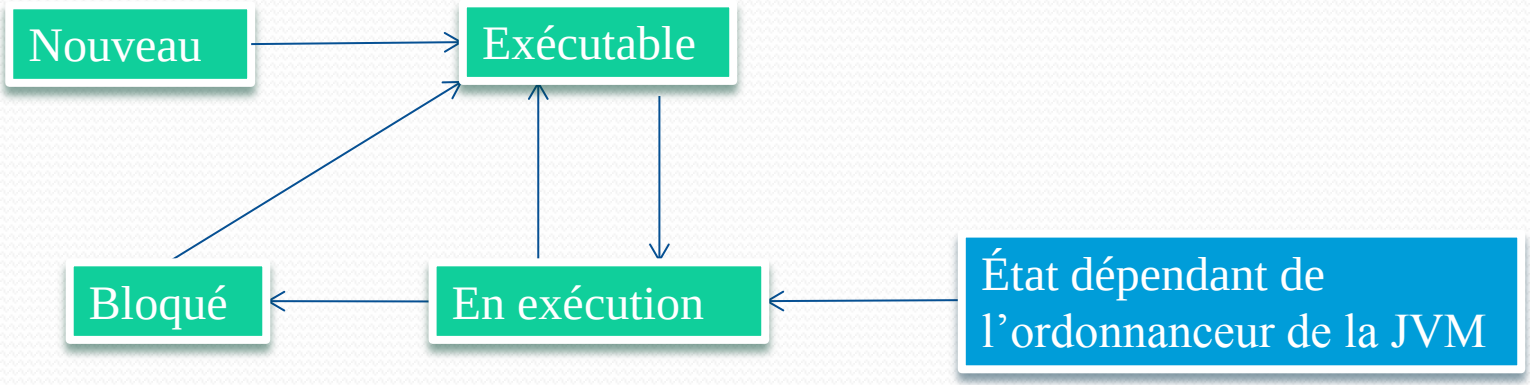
L'extension du fichier n'est pas obligatoire et n'a aucune importance

Thread

- Exécution (réellement ?) parallèle
- Tâche : interface `Runnable`
- *Thread* : classe `Thread`
- Lier la tâche au *thread*
- *Deadlock* (interblocage) :
  - Mot clef `synchronized` (pour éviter les *deadlocks*)



Threads




```
Runnable r = new MyJob(); // On crée une tâche (de type Runnable).
Thread myObj = new Thread(r); // On crée le Thread (état : nouveau).
myObj.start(); // Cela crée une nouvelle pile (état : exécutable).
Thread.sleep(200); // "Dors" pendant 200 ms (état : bloqué).

public synchronized void meth1() { … } // Cette méthode ne sera exécutée
                                       // que par un Thread à la fois.
```

Générique (Java 5)

- Polymorphisme paramétrique de type
- Comportement unique pour des types polymorphes
- **Un peu** comme les *templates* C++ :
  - Une seule copie du code : compilé une fois pour toutes
- Notation : <Type1>, <Type2, Type3>, etc.

Classes typées à la compilation

```
MyClass<String> myObj = new MyClass<String>();

public class MyList<B extends A, C>
MyList<MyClass, Date> list = new MyList<MyClass, Date>();
```

Génériques (Java 5)

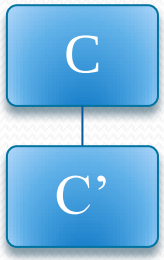
- *Wildcard* : ?

```
void myMeth1(List<? extends MyClass> a) {
    for(MyClass p : a) {
        myMeth12(p);
    }
}
```

- Variance (limite de portée) : &

```
final class MyClass<A extends Comparable<A> & Cloneable<A>,
                    B extends Comparable<B> & Cloneable<B>>
                    implements Comparable<MyClass<A, B>>,
                               Cloneable<MyClass<A,B>> {
    …
}
```

Si C' hérite d'une classe C et G est un générique de paramètre T alors : il est **faux** de dire que G<C'> hérite de G<C>



Il ne peut y avoir que des interfaces après le premier &

Collections (Java 2)

- Désavantages d'un tableau :
  - Taille statique
  - Recherche lente (exhaustive)
  - Pas de *pattern* de déplacement dans les éléments
- API Java :
  - Collection (interface `Collection`) :
    - Généricité et références (n'importe quelles références objets)
    - Opérations optimisées et communes
    - Itérateurs (parcourent les éléments un à un sans problème de type)
  - Tableau dynamique : `ArrayList`
  - Liste : `LinkedList`
  - Ensemble : `HashSet`, `TreeSet`

Itérateurs

- Monodirectionnel : interface `Iterator`
  - Toutes les collections en ont un :

```
// C'est une collection : on récupère son itérateur.
Iterator iter = c.iterator();
while (iter.hasNext()) {
    MyClass o = iter.next();
}
```

- Bidirectionnel : interface `ListIterator` (dérive de `Iterator`)
  - Listes et tableaux dynamiques uniquement
  - Deux sens

```
ListIterator iter = c.listIterator();
while (iter.hasPrevious()) {
    MyClass o = iter.previous();
}
```

Exemples de collections

- `LinkedList` (liste doublement chaînée)
- `ArrayList` (à la place de `Vector` qui est déprécié) :
  - Encapsulation du tableau avec une taille dynamique

```
ArrayList<String> arrList = new ArrayList<String>();
arrList.add("toto"); // Valide.
arrList.add(new String ("tata")); // Valide.
arrList.add(10); // Non valide ...
```

- `HashSet` :
  - Permet des éléments identiques
  - Prévoit la redéfinition des méthodes :
    - `hashCode()` : ordonnancer les éléments
    - `equals(...)`

Ellipse : varargs (Java 5)

- Nombre indéfini de valeurs de même type en paramètre
- Traitée comme un tableau
- Deux manières :
  - Avec un tableau (éventuellement vide)
  - Avec un ensemble de paramètres
- Placée en dernier dans la liste des paramètres
- En cas de surcharge de méthode, la méthode contenant l'ellipse a la **priorité la plus faible**

```
public meth1(Type... tab) { … }
…
int[] t = {1, 2, 3, 4, 5};
meth1(t); // Envoyé comme un tableau.
meth1(1, 2, 3, 4, 5); // Envoyé comme un ensemble de paramètres.
```

Utilisation de l'ellipse : …

Javadoc

- Outil standard pour créer une **documentation** d'API
- Génération automatique en HTML
- Utilisation (≠ commentaire `/* */`) :
  - Première ligne : uniquement `/**`
  - Lignes suivantes : un espace suivi de `*`
  - Dernière ligne : un espace suivi uniquement de `*/`
  - L'entité documentée est précédée par son commentaire
  - Tags prédéfinis

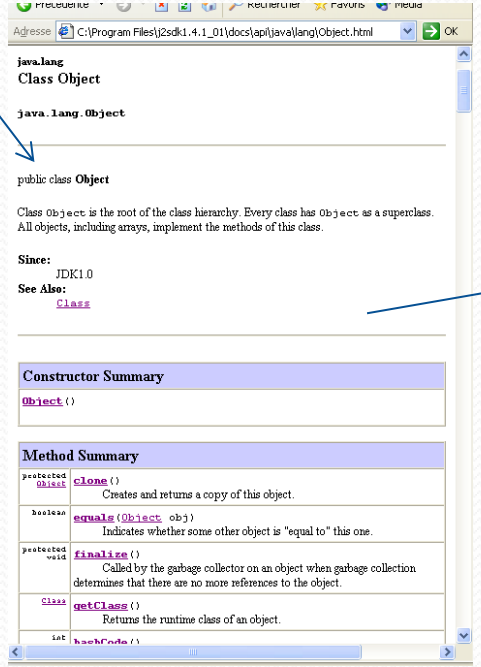
Bonne utilisation : expliquer n'est pas traduire !

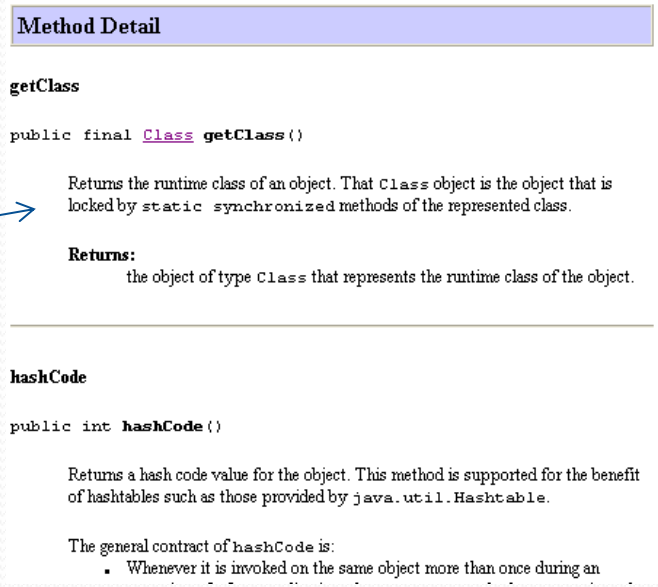
Javadoc : principaux tags

- `@author` : nom du développeur
- `@version` : version d'une classe/méthode
- `@param` : définit un paramètre de méthode :
  requis pour chaque paramètre
- `@since` : version du JDK de l'apparition de la classe/méthode
- `@return` : valeur de retour
- `@throws` : classe de l'exception et conditions de lancement
- `@deprecated` : marque la méthode comme dépréciée
- `@see` : référence croisée avec un autre élément

Exemple de Javadoc

```
/**
 * Valide un mouvement de jeu d'Échecs.
 * @param beginCol Colonne de la case de départ
 * @param beginRow Ligne de la case de départ
 * @param endCol   Colonne de la case de destination
 * @param endRow   Ligne de la case de destination
 * @return vrai(true) si le mouvement d'échec est valide ou faux(false) sinon
 */
```





JAR/WAR

- Formats de fichier
- JAR (extension `.jar`) :
  - Outil d'archivage du *bytecode* et des **métadonnées**
    - Fichier *manifest* : `MANIFEST.MF`

```
Manifest-Version: 1.0
Created-By: 1.4.1_01 (Sun Microsystems Inc.)
Main-class: HelloWorld
```

Fichier `MANIFEST.MF`

Classe principale à exécuter

  - On peut lire/utiliser le contenu d'un JAR
- WAR (extension `.war`) :
  - Assemblage de JAR pour une application Web
  - Utilisé pour un déploiement sur un serveur d'application

À savoir

- Interface `Cloneable`, permet de disposer de la méthode :

```
protected Object clone() { … }
```

- `enum` :

```
public enum Animal {KANGAROO, TIGRE, DOG, SNAKE, CAT, … };
```

- `instanceOf` :

A utiliser avec parcimonie

```
if (myObj instanceOf MyClass) {
    myObj2 = (MyClass) myObj; // Downcasting.
}
```

Bonnes pratiques

- Traitez toutes les exceptions susceptibles d'être lancées
- Faites attention à ne pas créer de *deadlock*
- Attention à l'héritage d'un générique

Aller plus loin

- Mot clef `volatile`
- Métaprogrammation par annotation
- Synchronisation de haut niveau (API de concurrence)
- API de management
- Gestion de flux standards : classe `Scanner`

Liens

- Document électronique :
  - http://nicolas.baudru.perso.esil.univmed.fr/Enseignement/enseignement.html
- Documents classiques :
  - Livre :
    - Claude Delannoy. *Programmer en Java 2ème édition.*
  - Cours :
    - Francis Jambon. *Programmation orientée application au langage Java*

Crédits

Auteur

Mickaël Martin Nevot
mmartin.nevot@gmail.com

Carte de visite électronique

Relecteurs

Cours en ligne sur : www.mickael-martin-nevot.com